10.1109/AHS.2019.00010

https://doi.org/10.24355/dbbs.084-202105041425-0

Demonstrating Controlled Change for Autonomous Space Vehicles

Alexander Dörflinger, Mark Albers, Björn Fiethe, Harald Michalik
Mischa Möstl, Johannes Schlatow, Rolf Ernst
Institute of Computer and Network Engineering (IDA)
Technische Universität Braunschweig
Braunschweig, Germany
{doerflinger, albers, fiethe, michalik, moestl, schlatow, ernst}@ida.ing.tu-bs.de

***Abstract*—Recent research discusses concepts of infield changes to overcome the drawbacks of conventional lab-based system design processes. In this paper, we evaluate the concept of controlled change by applying it to a demonstration of a potential future space exploration scenario with mobile robots. The robots are capable of executing several image computations for exploration, object detection and pose estimation, which can be allocated to both FPGA- and processor resources of a System-on-Chip. The demonstrator addresses three scenarios which cover application-, environment-, and platform change. The system adapts itself to any of the named changes. This capability can increase the autonomy of future space missions. Exemplary, the demonstrator executes adaption of applications during operation to fulfill the mission goals, adaption of reliability under changing environment conditions, and adaption to sensor failure.**

***Index Terms*—Autonomous Systems, Exploration, Object Detection, Pose Estimation, Reliability, Self-adaption**

I. Introduction

Large communication latencies between control station and space vehicles impede human supervising and decision making. Hence, future space missions strive for a high degree of autonomy. Autonomy is obtained through self-awareness and self-adaption. For self-awareness, a system requires knowledge about its current state, feasible actions and incidental effects, and the environment. Any change of one of these parameters may require the system to react and self-adapt to the new conditions. This adaption can be obtained through changing the system platform and/or system applications.

The state-of-the-art lab-based deployment process of system configurations proves the adherence of both functional and non-functional requirements on multiple integration levels. However, it strongly limits the adaption process to system configurations that have been considered and verified during the design time. An automated deployment of configurations provides new design space and adaption possibilities. Still, changing a system configuration has to be controlled in order to obviate invalid solutions. In particular, platform and application changes need to be checked for violations of non-functional requirements such as real-time, safety and reliability. This check can be executed in a model domain before deploying a new system configuration to the execution domain.

Current research discusses such controlled change [1]. Within the Controlling Concurrent Change project[1], several tools and methods for a supervised system adaption have been developed. This paper validates the concept by applying the mechanisms to a demonstrator of a potential future planetary surface exploration, consisting of two or more robots. Three change scenarios, covering application-, environment-, and platform change, are evaluated:

In a first scenario, the adaption of operation modes on a robot assists to achieve a superordinate goal. In the given demonstrator, a robot autonomously explores its environment and searches for objects of interest. Once a potential object has been detected, the robot changes its operation mode for approach, pose estimation, and possibly manipulation of the object.

A second scenario improves availability by adaption of reliability as a consequence of changing environment conditions. Single Event Effects (SEEs) caused through radiation particles are emulated in the demonstrator. Redundancy such as TMR on component level reduces processing performance, but prevents malfunction. The demonstrator has the capability to deploy both spatial and temporal redundancy concepts, which allows an accurate balancing between performance and reliability. Monitoring a minimum reliability goal of deployed system configurations enables the robot to adapt to changing solar weather conditions and particle flux intensities.

A third scenario demonstrates the adaption to sensor failure as an autonomous system reaction to degeneration. Missing sensor data, needed by one or more functions, will be restored through secondary sensor inputs if possible. Once such a recovery fails, functions can be redistributed between multiple robots to collaboratively accomplish the mission goal.

The rest of this paper is organized as follows. Section II introduces the robot hardware platform that will host applications described in Section III. The state-of-the-art process of mapping applications to a given platform is described in Section IV, and Section V provides a solution to overcome current limitations. Three change scenarios are discussed exemplary in Section VI.

[1] http://ccc-project.org/

https://doi.org/10.24355/dbbs.084-202105041425-0

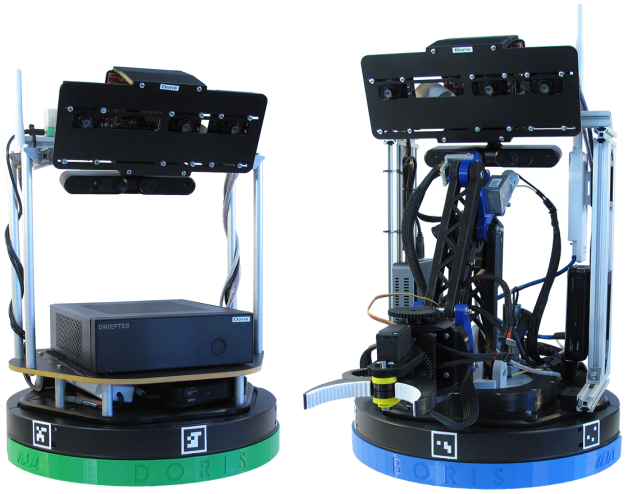

Fig. 1. DORIS and BORIS.

II. Hardware Platform

For demonstration purposes, a battery powered robot platform based on turtle bots has been developed. The platform is capable of fulfilling the tasks environment exploration, object detection, object pose estimation, and object manipulation. Two robot variants are pictured in Fig. 1: While the Demonstrator Of Reconfigurable Integrated Systems (DORIS) on the left is specialized for high-performance image computation, BORIS is capable of moving objects with its gripper arm. Of course, the mechanical and electrical setup of the robots is not adequate for use in space. Still, the general demonstrator architecture allows validating novel methods and mechanisms for autonomous space systems.

Fig. 2 depicts the platform architecture. A stereo camera subsystem is mounted at the very top of each robot. It contains in total three CMOS image sensors[2], which are connected to a Xilinx Zynq-7000 SoC[3]. Depending on the operation mode, different cameras will be activated. A single camera suffices for object detection. For pose estimation based on stereo-vision, the arrangement of cameras allows to choose a baseline of $64\,\mathrm{mm}$ (left-middle), $122\,\mathrm{mm}$ (middle-right), or $186\,\mathrm{mm}$ (left-right). While a camera pair with a wide baseline is selected for objects far away, a camera pair with a short baseline yields optimum results for nearby objects. The Zynq-7000 SoC contains an FPGA fabric connected to an ARM processor. This allows to efficiently accelerate several image computation steps by executing them in the FPGA part of the SoC.

The main controller is an Intel processor, which can be selected from two variants. The optional gripper arm for object manipulation reduces the remaining power and weight budget for the main controller, limiting the processor configuration. Hence, a robot can be specialized either for high image computation power, using an Intel PC[4], or object manipulation, using an Intel NUC[5]. The main controller supervises and configures the stereo camera controller and computes the targeted trajectory. An additional camera with depth sensor is connected to the main controller and supports the environment mapping.

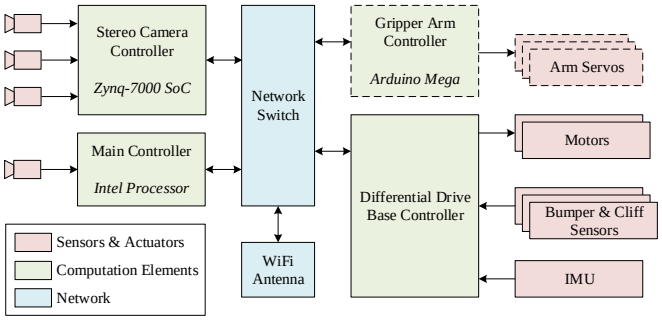

Fig. 2. DORIS / BORIS Platform Architecture.

The differential drive base controller collects sensor data from three bumper- and three cliff sensors and an IMU. This sensor information is used to limit the robot's movement options to a safe subset which mitigates collisions and obviates fall. Within this limitation, the trajectory proposed by the main controller will be followed as close as possible. The differential drive base controller generates the appropriate motor commands.

Robots specialized for object manipulation are equipped with a gripper arm. Several servo- and DC motors with magnetic rotary position sensors are connected to the gripper arm controller[6]. With six rotational joints, the end effector can move in all six degrees of freedom. It can reach a distance of approx. $550\,\mathrm{mm}$ from the robot's midpoint on the floor and a hight of approx. $440\,\mathrm{mm}$. A high friction inlay on the gripper improves grasping of slippery objects.

The four computation elements are connected with each other via a network switch. A WiFi antenna is mounted within the robot's mechanical structure, which enables communication between multiple robots.

The robots are equipped with several further features, e.g.: Unambiguous 4x4 markers are distributed evenly around the drive base and allow the robots to determine their relative position and orientation. In order to indicate different robot states, an RGB status light is mounted on top of the robot frame.

III. Application

The demonstration is based on a commonly imaging and stereo-vision use case. Multiple mobile robots search for specific objects while exploring an unknown environment and building a map of it. Once an object of interest has been detected, it will be analyzed and possibly manipulated. Fig. 3

[2]Aptina MT90M031 1/3-Inch Sensor
[3]Avnet MicroZed Development Board with a XC7Z020

[4]Intel Core i5 6400
[5]Intel NUC Kit NUC7i5BNK
[6]Arduino Mega 2560 Rev3

https://doi.org/10.24355/dbbs.084-202105041425-0

depicts a simplified state machine of the mission process flow. During regular execution, each robot performs this procedure on its own. In case of failures, collaborations are possible to facilitate fulfillment of the mission goal, which results in an interleaving of state machines. Each robot starts constructing a map of the unknown environment. Simultaneously, a robot keeps track of its location within the map using a Simultaneous Localization And Map (SLAM) algorithm. At the same time, a simple object detection algorithm is executed, which processes single RGB images and reports possible detections of objects being searched for. Once a potential object of interest has been found, the robot determines the object position, and approaches it. Subsequently, the object pose is estimated. The pose estimation algorithm uses two images of the stereo camera as input data. It performs a stereo matching to create a point cloud and fits templates of objects searched for into it. With an object's pose being determined, a robot equipped with a gripper arm can now manipulate the object. The object pose is then stored in the robot's map and the same procedure is performed for the next object until all objects of interest have been located.

As such robotic applications have strict requirements regarding the utilized operating system in matters of safety and reliability, the microkernel based Genode OS[7] enforcing a strong isolation between software components is used for the adaptable parts of this application. Genode OS organizes the adaptable parts as a tree of processes, where child processes are created out of the resources of their respective parent. A parent fully defines the virtual environment of its children. Each parent maintains full control over its arbitrary structured subsystems and defines their inter-relationship, for example by selectively permitting communication between them or by assigning physical resources. Some non-adaptable legacy parts, such as SLAM and navigation, rely on Robot Operating System (ROS)[8], a robotic meta-operating system, and are so far excluded from the adaption process.

[7]https://genode.org/

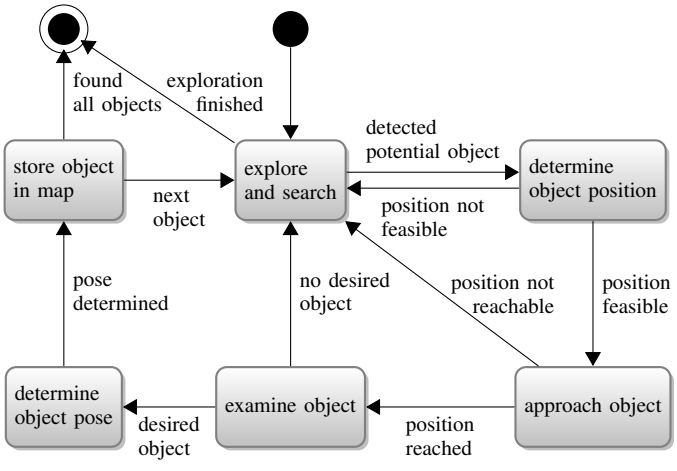

Fig. 3. State Machine of the Demonstration Scenario.

In the following, some details of the different operation modes are described. A function layer for each mode depicts coarse process flows. The function layer abstracts blocks for which no further details are given at this level. For each block there may exist different implementations, e.g. for hardware and software execution. For simplicity, only the image processing pipelines needed for object detection and pose estimation are shown.

A. Object Detection

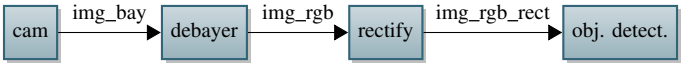

Fig. 4. Function Layer of Object Detection.

The *debayer* block converts Bayer images received from the *cam* block to RGB images. Camera distortions are removed using a *rectify* block and are analyzed by the *object detection* block. Fig. 4 illustrates the function layer of this scenario. Edges indicate a functional dependency between blocks and are labeled with the corresponding interface.

B. Pose Estimation

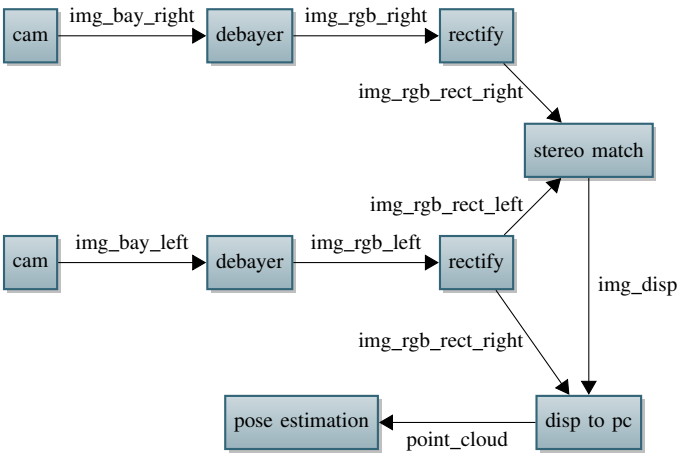

Fig. 5. Function Layer of Object Pose Estimation.

As depicted in Fig. 5, two *cam* blocks create synchronized left and right images in raw Bayer form. Two *debayer* blocks execute the Bayer to RGB conversion for left and right images respectively. The *rectify* blocks transform both images into a common coordinate system. Subsequently, the *stereo match* block calculates a disparity map from rectified left and right images. The *disp to pointcloud* block converts the disparity values to 3D points. Templates of the objects searched for are then fitted into the point cloud by the *pose estimation* block to determine their poses.

[8]http://www.ros.org/

https://doi.org/10.24355/dbbs.084-202105041425-0

C. Hardware Acceleration

The image processing during object detection and pose estimation contains computationally intensive algorithms. Their execution on a processor consumes a rather high energy budget and yields only low frame rates. However, operations on large datasets such as images have a high potential of parallelization. Therefore, hardware components for an accelerated execution within the FPGA fabric of the Zynq-7000 SoC have been implemented for several computation steps, here to name *debayer*, *rectify*, *stereo match*, and *disp to pointcloud*. Some computations yield high acceleration rates when executed in hardware, which improves the overall performance and reduces the power consumption. The resources within the FPGA are limited and hence the number of hardware accelerated components. For a even more efficient use of the FPGA, the Dynamic Partial Reconfiguration (DPR) capability has been implemented on the demonstrator. It allows to time-share FPGA resources between different hardware components, e.g., the *debayer* and *rectify* can be executed in a first time step, while the *stereo match* will be loaded into the FPGA and executed in a second time step. Reconfiguration times range from $8\,\mathrm{ms}$ to $18\,\mathrm{ms}$, depending on the size of the hardware component within the FPGA. Execution times of the hardware modules range from $36\,\mathrm{ms}$ to $528\,\mathrm{ms}$. In contrast to a context switch on a processor, the reconfiguration time on the FPGA is not negligible. Still, the high acceleration rates (e.g., the *disp to pointcloud* component executes 32 times faster in hardware than in software) compensate this drawback. More details on system reconfiguration and the DPR capability under Genode OS used in this demonstration are given in [2].

IV. State-of-the-Art and Related Work

The V-model is the state-of-the-art process model in many domains that involve critical systems, such as space [3], automotive [4] and other industrial equipment [5]. In this process model the descending branch of the V contains the specification of the system top-down, i.e. from system level requirements to requirements of atomic elements. The ascending branch contains verification and test of these requirements. Implementation is performed at the bottom tip of the V. If development processes strictly adhere to this model, updates to existing systems become a laborious process as the process has to be reset to the point on the V where requirements diverge. In strict cases, a minimal change to one requirement implies to reiterate the entire ascending branch.

For systems with high degree of autonomy, all possible configurations would require testing and verification according to the V-process model. This calls for an automation of parts of the process. We do not strive for autonomous requirement specification at higher levels of system design. However, we do want to automate the design-space exploration and subsequent configuration based on higher-level requirements and objectives of a space vehicle.

This line of work can be summarized under the term *automated design-space exploration*. There are many approaches that try to model the design space in order to automatically perform architecture optimization: Terzimehic et al. [6] propose Satisfiability Modulo Theories (SMT) for formulating constraints and objectives derived from hardware and software annotations. ProMARTES [7] is an approach that combines profiling, analysis and simulation for architecture performance optimization using Genetic Algorithms (GAs). Eder et al. [8] use a dedicated domain-specific modeling language to formulate memory, safety, cost, energy and bandwidth constraints for the mapping problem, i.e. the allocation of software components to hardware components. The main limitation of these approaches is that the software composition is already known and fixed. There are approaches that explicitly address the software composition problem using constraint-based methods such as SMT [9] or Mixed Integer Linear Programming (MILP) [10], however, they do not consider cases in which the number of component varies (cf. Fig. 8).

V. Multi Change Controller

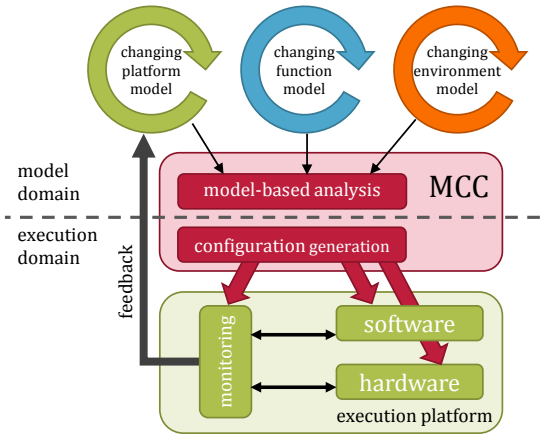

Fig. 6. The Multi-Change Controller configures the execution platform.

The Multi-Change Controller (MCC) developed by us manages the configuration of the execution platform in order to allow infield adaptation to intended changes or unforeseen events [1]. Instead of deploying a static configuration for every operation mode, the MCC equips the robot with the capability to generate and deploy configurations dynamically at runtime. A configuration is a composition of interconnected components that use certain interfaces to exchange data. A component corresponds either to a binary that can be executed in the software system or a bitstream that can be executed in the FPGA fabric. As illustrated by Fig. 6, the inputs of the MCC are a changing platform model, a changing function model and a changing environment model. The platform model specifies the structure and state of the software system (e.g. runtime environment, operating system) and hardware system (e.g. processing units, networks, sensors, actors). Changes such as failing hardware will be detected by runtime monitoring ([11], [12]) that observes the current state of the execution platform. The function model specifies the function architecture (cf. Fig. 4&5), i.e. the intended functionality of the robot. This also includes constraints such as performance

https://doi.org/10.24355/dbbs.084-202105041425-0

requirements (e.g. latency, frame rate) and reliability requirements. The environment model provides information about the operating conditions (e.g. temperature, radiation intensity) that are important for the execution platform as they impact essential properties of the processing system and actuator capabilities of the robot (e.g. drive speed).

In order to adaptively configure the system according to the given state of platform, function and environment, the MCC automatically performs a model-based integration, i.e. it tries to find a configuration that matches the current state of the input models. This is achieved by a stepwise enrichment and synthesis of model layers, starting with the platform- and implementation-independent function architecture as the first model layer. The MCC will derive several other model layers until a concrete platform- and implementation-specific model of the execution platform has been found that satisfies all requirements. As this may potentially lead to an exhaustive design-space exploration, we employ a heuristic approach that emulates the design decisions that would be performed in a manual integration process by a human engineer.

The incremental enrichment and synthesis of model layers is performed by a predefined sequence of *parameter decisions*, *transformations* and *checks*. In the scope of this paper, we focus on the function layer, the component layer and the task layer. The *function layer* represents the given function architecture which is an abstract specification of the intended functionality. The *component layer* represents the component architecture that specifies how the functions are implemented by a set of interconnected hardware/software components. The *task layer* provides a timing view on the system, as it models the communication between the components [13], which is required for performance analysis.

Fig. 7 schematically illustrates the process of creating a component layer from a function layer and a task layer from a component layer. First, parameter decisions are performed on the function layer before the transformation into the component layer can be conducted. The result is checked for consistency, invariants, etc. and, if failed, the causing decision is revised by rolling back the sequence of operations up to that point. If the check succeeds, the necessary parameter decisions are conducted on the component layer and the transformation into the task layer is performed. This process is continued until a configuration could be generated from the component and task layer (or until there is no parameter decision any more that can be revised). In case a new configuration could not be generated, the requested function layer cannot be used and must be replaced with an alternative or fallback. In the meantime, the system will continue operating with the current configuration.

Parameter decisions are e.g. mapping of functions to processing units, assignment of resources, the selection of components for a given function, or the synthesis of a hardware/software co-schedule [14].

Most parameter decisions build the basis for the *transformations* as illustrated in Fig. 8. The figure depicts a function layer with a function A and a function B. For function B, there

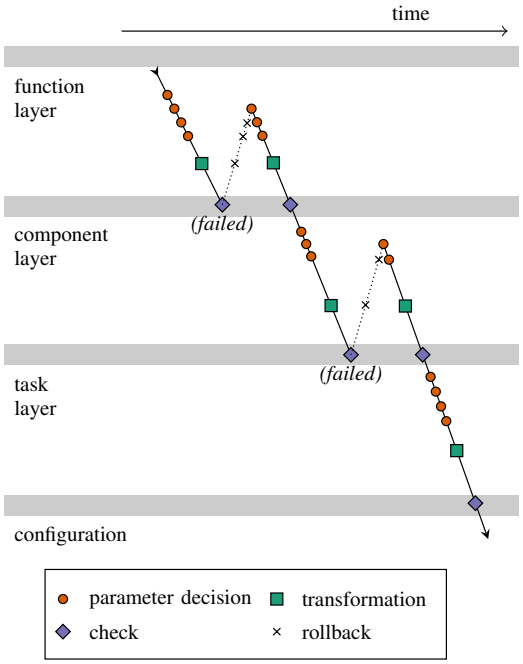

Fig. 7. The configuration is found by performing parameter decisions, transformations and checks.

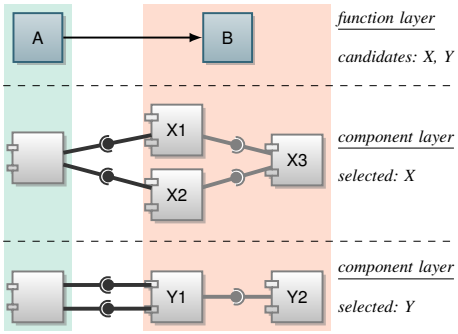

Fig. 8. A function B is transformed into components X1, X2, X3 or components Y1, Y2.

are two implementation candidates: X and Y. Implementation X consists of three components (X1, X2, X3) whereas implementation Y consists of two components (Y1, Y2) such that the component layer will look differently depending on the selected implementation. Different implementations typically vary in their resource requirements and/or the achievable performance or reliability. A more detailed account of what decisions and transformations must be performed to derive a component architecture from a function architecture is given in [15].

Checks are admission tests of resulting layer(s) that are performed in order to provide assurance of requirements. Typical admission tests are response-time analysis [13], safety analysis [16], or reliability analysis.

https://doi.org/10.24355/dbbs.084-202105041425-0

VI. Change Scenarios

The MCC described in Section V allows autonomous deployment of new configurations onto the execution platform. This ability will be used in the following scenarios, which demonstrate the benefit of the new approach. The scenarios cover change in all three input models (application, environment, and platform) of the MCC. Admission tests will be executed to check performance and reliability constraints respectively.

A. Application Change

Typically, an application change is an intended change and can be well prepared during a lab-based development phase using state-of-the-art integration methods. Examples for such intended change are in-field feature updates for embedded systems with long lifetimes. This is a common method already being applied for space probes, as many updates are deployed only after launch.

Another example for intended change is a time-multiplexed execution of different applications on the same hardware platform, which will be discussed in the presented demonstration. Likewise, the described procedure can be applied to simplify and support updatability.

To fulfill the overall mission goal, a robot has to switch between multiple operation modes. The operation modes *Object Detection* and *Pose Estimation* consist of different function layers as described in Section III. Both modes will be executed sequentially on the same platform. In the demonstrator, a function layer for the image processing is deposited for each operation mode of the state machine in Fig. 3. On transition from one operation mode to another, the MCC is fed with an updated function layer and the reconfiguration process will be triggered. The MCC searches for a valid system configuration for the new function layer and iterates through the layers discussed in Section V. Under normal conditions (no harsh environment conditions, no platform failures) the MCC selects standard components for the elements of the *Object Detection* function layer and maps them to the platform resources as depicted in Fig. 9. Communication modules that allow to connect components, such as Direct Memory Access (DMA) Controllers and Network Interface Controllers (NICs), are automatically instantiated in the component layer. Note that the resulting component instantiation on the platform is only one example of multiple possible solutions. The DPR capability, that has been implemented for the demonstrator, allows to schedule tasks on the FPGA fabric similarly as on a processor, as FPGA resources can be shared between tasks in a time-partitioned manner. The decision of the MCC to execute the *debayer* and *rectify* hardware accelerated in the FPGA fabric of the Zynq-7000 SoC is based on the HW/SW task scheduling [14] and results in a high frame rate of processed images of 3.0 fps.

The function layer of the *Object Pose Estimation* is handled similarly. However, due to the higher graph complexity, the component selection and mapping is a more sophisticated process. For a more even distribution of loads on platform resources, some components that could be hardware accelerated are still executed in software. E.g., the ARM processor in the Zynq-7000 SoC is running idle during the *Object Detection* mode, while three components are being executed within the FPGA part (see Fig. 9). During the *Pose Estimation* mode, additional components are mapped to the FPGA, forcing the *debayer* to move to the ARM processor for a software execution.

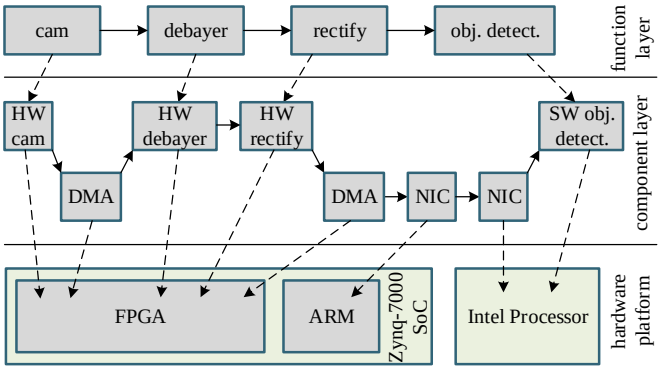

Fig. 9. Component Selection and Mapping for the Object Detection.

B. Environment Change

In contrast to application change, an environment change is not intended by operation and not always predictable beforehand. Typically, the range of environment conditions during operation (e.g. min/max temperature) is evaluated and defined at design time. However, the actual development of environmental conditions is at least partly unpredictable. A permanent adaption of a system to the current environment can improve several aspects such as average performance (e.g., optimum bandwidth use for different levels of electromagnetic noise) and safety (e.g., adapted speed to different weather conditions). For autonomous systems, self-adaption to such environmental changes is essential. The MCC can be used as an instrument to react to environment change and execute the self-adaption.

For systems operating in space, radiation intensity is an important environment factor, hence it will be discussed exemplary in the demonstration. Once a radiation particle penetrates silicon, it can cause SEEs in the circuitry, which can result in erroneous behavior or even system failure. Soft error induced system failures can be statically reduced through costly replication of circuits or avoidance of error-sensitive acceleration components such as FPGA resources. However, this often results in poor resource usage when operating in normal conditions. Hardening the system against soft errors to achieve a certain reliability might only be necessary during short time intervals with harsh radiation conditions and a frequent occurrence of soft errors. By constantly adapting a system to its current environment, the average performance can be improved [17].

https://doi.org/10.24355/dbbs.084-202105041425-0

This type of adaption is analyzed for the pose estimation phase of the DORIS demonstrator in an exemplary manner. In total 128 configurations with varying compositions of (TMR-) HW and SW components can execute the pose estimation function layer. E.g., the *HW debayer* component in Fig. 9 can be replaced by a *TMR HW debayer*, or concatenated twice for temporal redundancy. With a reliability check, the MCC is able to rate each potential configuration regarding its reliability for a given particle flux. The component selection affects the resulting reliability, as it defines the number of utilized resources, which are exposed to radiation. Hereby, the reliability check takes varying soft error susceptibilities of different resources (memory, CPU, and FPGA fabric apportioned into configuration memory, block RAM, and registers) into account.

For the reliability check, the intensity of the particle flux has to be known. We currently resort to external reporting such as the space weather forecast, yet system internal sensors that compute the particle flux based on the observed bit error rate in block RAM of the FPGA, are in principle possible [17].

Analyzing the behavior of the complete system under changing radiation conditions would require tests at radiation facilities. For the demonstrator, we resort to a soft error emulation through error injection within the FPGA part.

Once a configuration has been deployed successfully to the execution platform, its reliability needs to be reevaluated on environment changes. By suspending the reevaluation within an upper and lower bound of the particle flux, the computation overhead can be kept low. Once a reliability constraint fails after reevaluation, new configuration candidates are requested until a new valid solution has been found, which will be deployed to the platform.

C. Platform Change

Another drawback of long mission lifetimes is the danger of permanent failure of electronic components. Harsh radiation environments afflict sensor elements in particular. If possible, such isolated wearouts should not result in a failure of the whole mission.

In case of a sensor failure, the MCC first searches for a feasible configuration for the current operation mode, that omits the use of the failed component. This mechanism has been validated with loss of one or more cameras on DORIS. During exploration, the image information of one camera is being processed. If the currently running camera sensor fails, the MCC checks alternative configurations. As long as there is at least one camera sensor that still functions, a solution will be found ultimately. The same process is triggered for a camera loss during the pose estimation phase. However, this operation mode requires two input images, and one failing camera already implies operational restrictions as the number of selectable camera pair baselines is reduced. The robot is still capable of executing its task (e.g., by selecting a medium baseline instead of a small baseline), however the quality of the pose estimation might suffer.

To prepare for failures using the classical design process, one would have to provide and thoroughly test a configuration for each possible failure and combination of failures. This seems feasible for the demonstrator, which offers 3 different configurations for each operation mode allowing reaction to different camera failures. However, it is already a quite time-consuming and expensive process. Typically, today's embedded systems feature even higher complexity, which makes the state-of-the-art process of integration and test for each backup configuration difficult to handle due to the high number of possibilities. Alternatively, a typical approach is a post-failure manual reaction. Space vehicle with redundant hardware operate, until the primary circuitry fails. The space vehicle reports the failure, and an operating manager decides on an action such as powering up a redundant component. However, this contradicts autonomous space vehicle design. Additionally, the manual elimination of a failure lengthens the period of faulty operation, which may result in subsequent faults.

Those disadvantages are solved by applying the proposed autonomous adaption to failure. The design process is simplified, as the MCC adopts cumbersome steps of integration and test.

If no valid configuration exists for the current operation mode and platform restricted by failures, this typically results in irrecoverable mission failure. In the given demonstration, the MCC attempts one more recovery by triggering an agreement protocol [18] for a new task distribution between multiple robots. Once two cameras fail on a robot, it is not able to perform the object pose estimation on its own anymore. However, it can still explore unmapped areas and search for undetected objects. Hence, the tasks object detection and pose estimation can be re-distributed between the robots.

VII. Conclusion

In this paper, we presented a validation of controlled change concepts on a potential future planetary surface exploration demonstrator with mobile robots. The presented model-based integration process that is implemented in the MCC allows the autonomous system to become self-aware in the sense that the system is able to recognize its own state (platform, environment, and function), possible actions, and the result of these actions on itself and its operational goals. We showed, how the integrated MCC enabled our setup of mobile robots to self-adapt to various predicted and unpredictable situations with computing, verifying and executing both platform and application change. We believe that further reaching (technical) self-awareness based on technical models can pave the way for fully autonomous exploration scenarios.

Acknowledgment

This work is part of the DFG Research Group FOR 1800 "Controlling Concurrent Change". Funding for the Institute of Computer and Network Engineering (IDA) was provided under grant number MI 1172/3-1.

https://doi.org/10.24355/dbbs.084-202105041425-0

REFERENCES

[1] M. Möstl, J. Schlatow, R. Ernst, N. Dutt, A. Nassar, A. Rahmani, F. J. Kurdahi, T. Wild, A. Sadighi, and A. Herkersdorf, "Platform-centric self-awareness as a key enabler for controlling changes in cps," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1543–1567, Sep. 2018.

[2] A. Dörflinger, M. Albers, J. Schlatow, B. Fiethe, H. Michalik, P. Keldenich, and S. P. Fekete, "Hardware and software task scheduling for arm-fpga platforms," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Aug. 2018, pp. 66–73.

[3] *ECSS-M-ST-10C Space project managment - Project planning and implementation*, 1st ed., European Cooperation for Space Standardization - ECSS, Mar. 2009.

[4] *ISO 26262 - Road vehicles - Functional safety*, 2016th ed., Intern. Organization for Standardization - ISO, Apr. 2016.

[5] *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2nd ed., The International Electrotechnical Commission - IEC, Apr. 2010.

[6] T. Terzimehic, S. Voss, and M. Wenger, "Using design space exploration to calculate deployment configurations of IEC 61499-based systems," in *14th IEEE International Conference on Automation Science and Engineering, CASE 2018, Munich, Germany, August 20-24, 2018*, 2018, pp. 881–886.

[7] K. Triantafyllidis, W. Aslam, E. Bondarev, J. J. Lukkien, and P. H. de With, "Promartes: Accurate network and computation delay prediction for component-based distributed systems," *Journal of Systems and Software*, vol. 117, pp. 450 – 470, 2016.

[8] J. Eder, S. Zverlov, S. Voss, M. Khalil, and A. Ipatiov, "Bringing dse to life: Exploring the design space of an industrial automotive use case," *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 270–280, 2017.

[9] S. Peter and T. Givargis, "Component-based synthesis of embedded systems using satisfiability modulo theories," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 4, pp. 49:1–49:27, Sep. 2015.

[10] D. Kirov, P. Nuzzo, R. Passerone, and A. L. Sangiovanni-Vincentelli, "Archex: An extensible framework for the exploration of cyber-physical system architectures." in *Design Automation Conference*. ACM, 2017, pp. 31:1–31:6.

[11] M. Möstl, J. Schlatow, and R. Ernst, "Synthesis of monitors for networked systems with heterogeneous safety requirements," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2824–2834, Nov. 2018.

[12] J. Schlatow, M. Möstl, and R. Ernst, "Self-aware scheduling for mixed-criticality component-based systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, Apr. 2019.

[13] J. Schlatow and R. Ernst, "Response-time analysis for task chains in communicating threads," in *22nd IEEE Real-Time Embedded Technology & Applications Symposium (RTAS)*, Vienna, Austria, Apr. 2016.

[14] A. Dörflinger, M. Albers, B. Fiethe, and H. Michalik, "Hardware acceleration in genode os using dynamic partial reconfiguration," in *Architecture of Computing Systems – ARCS 2018*, M. Berekovic, R. Buchty, H. Hamann, D. Koch, and T. Pionteck, Eds. Cham: Springer International Publishing, 2018, pp. 283–293.

[15] J. Schlatow, M. Möstl, R. Ernst, M. Nolte, I. Jatzkowski, and M. Maurer, "Towards model-based integration of component-based automotive software systems," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, Oct. 2017, pp. 8425–8432.

[16] M. Möstl and R. Ernst, "Cross-layer dependency analysis with timing dependence graphs," in *Proceedings of the 55th Design Automation Conference (DAC)*, 2018.

[17] R. Glein, B. Schmidt, F. Rittner, J. Teich, and D. Ziener, "A self-adaptive seu mitigation system for fpgas with an internal block ram radiation particle sensor," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 251–258.

[18] W. Xu, M. Wegner, L. Wolf, and R. Kapitza, "Byzantine agreement service for cooperative wireless embedded systems," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Jun. 2017, pp. 10–15.

https://doi.org/10.24355/dbbs.084-202105041425-0